Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Problemas NP-Completos

24 de Noviembre 2023

Tobías Serpa
108266

Carolina Mauro
108294

Valentina Adelsflugel
108201

1. Objetivo

El objetivo de este trabajo práctico es obtener una composición óptima de un equipo de jugadores para un partido de fútbol amistoso, considerando las preferencias de diversos medios de comunicación. Este problema específico se formula como una instancia del Hitting-Set Problem, que consiste en determinar un conjunto mínimo de jugadores que satisfacen las demandas de la prensa. El trabajo propone demostrar que el Hitting-Set Problem pertenece a la clase NP, así como demostrar su condición de NP-Completo. Además, se busca desarrollar e implementar un algoritmo de Backtracking para obtener la solución óptima, contrastándolo con un modelo de programación lineal. Finalmente se plantea también una aproximación utilizando variables reales y redondeo, explorando su eficacia y comparándola con la solución exacta.

2. ¿Qué es NP y NP-Completo?

En el ámbito de la teoría de la complejidad computacional, los problemas se clasifican en diversas clases en función de su capacidad de resolución eficiente. El conjunto P abarca problemas que pueden resolverse de manera eficiente en tiempo polinomial mediante una máquina de Turing determinística. Por otro lado, la clase NP engloba problemas para los cuales existe un certificador eficiente, lo que implica que la solución puede ser validada en tiempo polinomial. Es decir, estos problemas pueden resolverse en tiempo polinomial por una máquina de Turing no determinística.

Un problema NP-completo es un tipo especial de problema en la teoría de la complejidad computacional. Un problema específico es NP-completo si pertenece a la clase NP (donde las soluciones pueden ser verificadas eficientemente en tiempo polinomial), y al mismo tiempo, cualquier problema en la clase NP puede reducirse de manera polinomial a él. En otras palabras, un problema NP-completo es tan difícil como los problemas más difíciles en NP, aunque aún no se haya demostrado que sea NP-completo.

3. ¿Qué es Programación Lineal?

Es una técnica de diseño (matemática) que permite resolver problemas de optimización de un sistema de ecuaciones lineal en varias variables.

Componentes:

- Variables continuas o enteras
- Ecuaciones e inecuaciones lineales que definen restricciones sobre esas variables
- Una función objetivo que buscamos maximizar o minimizar
- Un algoritmo que resuelve el modelo lineal

4. ¿Qué es un Algoritmo de Aproximación?

Estos algoritmos surgen de la pregunta ¿cómo diseñamos algoritmos para problemas en los que el tiempo polinomial es probablemente inconseguible?

Características:

- Diseñados para correr en tiempo polinomial
- Encuentran soluciones que garantizan estar cerca del óptimo
- Se puede demostrar que encontramos solución cerca del óptimo

- La solución óptima es muy difícil de calcular y aún así vamos a poder demostrar la cercanía de la solución aproximada

Hay varias metodologías de aproximación: Greedy, Pricing Method, Programación Lineal, Redondeos, Programación Entera y Programación Dinámica.

5. Resolución del trabajo

5.1. Demostrar que el Hitting-Set Problem se encuentra en NP

El Hitting-Set Problem dice que: dado un conjunto A de n elementos, m subconjuntos $B_1, B_2, ..., B_m$ de A y un número k, ¿existe un subconjunto C incluído en A con $|C| \leq k$ tal que C tenga al menos un elemento de cada B_i?

Para demostrar que este problema se encuentra en NP debemos ver si encontramos un verificador polinomial. ¿Cómo hacemos para validar si una solución C es correcta? Si contamos con los subconjuntos B_i, tenemos que asegurarnos que cada B_i tenga al menos una intersección con C. Esto se puede lograr con un algoritmo como el siguiente:

```
def verificar_solucion(B, C, k):
    if len(C) > k:
        return False
    satisface_Bi = [False] * len(B)

    for c in C:                         # O(#C=k)
        for i in range(len(B)):
            if c in B[i]:               # O(#B[i])
                satisface_Bi[i] = True

    return all(satisface_Bi)            # O(#B)
```

Como vemos es de orden polinomial porque tenemos un primer for que se ejecuta k veces (o incluso menos dado que el problema plantea buscar $|C| \leq k$) y dentro de este, se van a ejecutar m sentencias if siendo cada una un ciclo (implícito) de orden igual a la cantidad de elementos del respectivo subconjunto B_i. Por lo tanto, la complejidad quedaría:

O(k) * [O(B1) + ... + O(Bm)] = O(k²) dado que en el peor caso, hay un subconjunto B_i que contiene a todos los elementos de la solución.

Como consecuencia, el Hitting-Set Problem se encuentra en NP.

5.2. Demostrar que el Hitting-Set Problem es, en efecto, un problema NP-Completo.

Habiendo demostrado que el problema está en NP, podemos avanzar a demostrar que es NP-Completo. Para poder hacer esto debemos reducir polinomialmente algún problema NP-Completo ya conocido a Hitting-Set (HS). El problema que vamos a reducir es Vertex Cover. Vertex Cover consiste en, dado un grafo $G = (V, E)$, obtener el conjunto mínimo de vértices tales que por lo menos uno de los dos extremos de cada arista este en él. En su versión de decisión es encontrar el Vertex Cover de tamaño menor o igual a k.

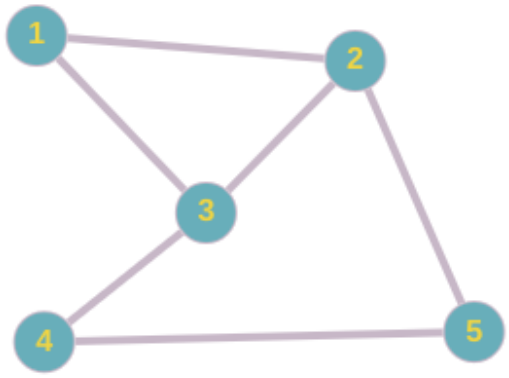
Para lograr la reducción:

- El conjunto de vértices V será equivalente al conjunto A de HS
- Cada subconjunto B_i en A será equivalente a $\{u, v\}$, una arista en el grafo G.
- La constante k será la misma para ambos problemas.

Entonces, si VC es el Vertex Cover G de tamaño k, esto implica que para cada arista $\{u, v\}$, u o v pertenece al VC. Por lo tanto, VC forma el HS porque todos los subconjuntos formarán una intersección con los vértices en VC.

Ahora bien, dado que el HS interseca cada subconjunto de A, al menos uno de los extremos de cada arista $\{u, v\}$ debe pertenecer a la solución. Por lo tanto, abarca al menos un vértice para cada arista, formando así el VC.

Veamos un ejemplo. Supongamos que tenemos el siguiente grafo:



Queremos un Vertex Cover de tamaño $k = 3$. Esto es $VC = \{2, 3, 4\}$.

¿Cómo reducimos esto a Hitting-Set?

- Necesitamos el conjunto A de n elementos.

  Esto es, el conjunto de vértices del grafo: $A = \{1, 2, 3, 4, 5\}$.

- Necesitamos m subconjuntos B en A.

  Esto es en el grafo, cada par de vértices conectados entre sí: $B_i = [\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 5\}, \{4, 5\}, \{3, 4\}]$

Resolver el HS Problem dados A y B es encontrar el subconjunto C de tamaño 3 incluído en A tal que C tenga al menos un elemento de cada B_i. En este caso, $C = \{2, 3, 4\}$ cumple, porque todos los subconjuntos B tienen al menos un elemento en común, y además es minimal. Podemos ver que la solución coincide con el Vertex Cover, siendo éste reducible a HS.

El Hitting-Set Problem es entonces NP-Completo.

5.3. Algoritmo de Backtracking

Realizamos un algoritmo que, por backtracking, obtiene la solución óptima al problema. Generamos también sets de datos propios para corroborar su correctitud y tomamos mediciones de tiempos.

Para este algoritmo, lo que hacemos es probar todas las combinaciones posibles de jugadores que generan nuevas soluciones. En búsqueda de evitar analizar combinaciones que no arrojan nuevas soluciones, generamos diferentes condiciones de corte a lo largo del código.

5.4. Condiciones de Corte y Optimizaciones

En primer lugar, antes de buscar solucionar el problema, lo que hacemos es ordenar los pedidos de los periodistas de los más cortos a los más largos. Hacemos esto debido a que, por ejemplo, si

existe un periodista que en su pedido solo pide un jugador, y a su vez es el único periodista que pide a ese jugador, sabemos que ese jugador tiene que estar obligatoriamente en nuestra solución. Es por eso que buscamos empezar cumpliendo con los periodistas que limitan nuestras soluciones. Gracias a esto, exploramos muchas menos soluciones posibles.

```
    pedidos_ordenados = sorted(pedidos_de_prensa, key= len)
```

Luego la solución es algo primitiva, ya que sólo es ir agregando jugadores de los distintos pedidos, hasta que se cumpla el objetivo, lo cual verificamos con la siguiente función:

```
    def verificar_solucion_para_Scaloni(B, C):
    for subconjunto in B:
        if not any(elemento in C for elemento in subconjunto):
            return False
    return True
```

Nos resulta interesante detallar las condiciones de corte que utilizamos para evitar analizar ramas que no devolverán nuevas soluciones. En primer lugar, chequeamos que nuestra combinacion actual de jugadores, no sea igual al largo de nuestra solución, ya que de ser así no tendría logica seguir agregando jugadores, debido a que cualquier solución encontrada poseería más elementos que nuestra solución.

```
    if(largo_solucion > 0 and cantidad_de_jugadores == largo_solucion):
        return False
```

A continuación, antes de agregar un nuevo elemento, chequeamos que el pedido del periodista del cual vamos a agregar jugadores, no esté siendo abarcado en la combinación de jugadores que estamos probando (nuestra solucion actual).

```
    def satisfecho(pedido, solucion):
        return any(jugador in pedido for jugador in solucion)
```

De ya estar abarcado en nuestra solución el pedido del periodista, continuamos con el siguiente.

```
    if satisfecho(pedido_actual, solucion_actual):
        return conjunto_minimo_para_Scaloni(pedidos_de_prensa, solucion_actual, i +
     1, solucion)
```

5.5. Modelo de programación lineal entera

Mediante la programación lineal entera logramos crear un algoritmo que resuelve de manera óptima el problema de Scaloni. Sabemos que nuestro objetivo es minimizar la suma de los elementos del conjunto C, tal que se cumpla que para todo $B_i \in B$, $B_i \cap C \neq \emptyset$.

Para transformarlo a un problema que podemos resolver con la programación lineal entera, vamos a pensar como que cada jugador es una variable x_i la cual puede tomar valor 0 o 1. Entonces, para cada set (en este caso, para cada pedido de un periodista) se debe cumplir que $x_j = 1$ para algún j. Es decir que dadas n ecuaciones (n = cantidad de pedidos), cada una con m variables (que representan a los jugadores que cada periodista solicita), buscamos minimizar la suma de las variables, tal que se cumpla que para cada ecuación, al menos una de las variables es 1.

5.5.1. Implementación Algoritmo

Para implementar el algoritmo utilizamos la libreria *pulp* la cual incluye soporte para problemas de maximización y minimización con restricciones. En este caso al estar frente a un problema de minimización, definimos nuestro problema inicial como:

```
    prob = pulp.LpProblem("Scaloni", pulp.LpMinimize)
```

Luego debemos de definir las variables de nuestro problema. En ese caso lo x_j son los jugadores pedidos por la prensa.

```
    jugadores = obtener_todos_jugadores(pedidos_de_prensa)
    x = pulp.LpVariable.dicts("x", jugadores, lowBound=0, upBound=1, cat="Integer")
```

Una vez definido el problema y las variables debemos sumar las ecuaciones que determinan cómo se relacionan dichas variables. Inicialmente se define la función objetivo, que es la suma de todas las variables de decisión. El objetivo es minimizar la cantidad de jugadores en el conjunto solución. Luego se itera por cada pedido de la prensa y se establece una restricción para cada pedido. La restricción asegura que al menos uno de los jugadores del pedido esté en el conjunto solución.

```
    prob += pulp.lpSum([x[i] for i in jugadores])
    for pedido in pedidos_de_prensa:
        prob += pulp.lpSum([x[jugador] for jugador in pedido]) >= 1
```

Finalmente, resolvemos con el solucionador predeterminado y obtenemos la solución quedandonos con aquellas variables que estén encendidas.

```
    prob.solve()
    solucion = [x[i] for i in jugadores if x[i].value() == 1]
```

5.5.2. Mediciones

Para poder comparar la solución por pogramación lineal y la solución por Backtracking decidimos crear 6 set de datos donde hacemos enfacis en la minima cantidad de jugadores que cada periodista elije. Notamos que esta era la variable que hacia que nuestro algoritmo auemnte su complejidad temporal. Por esta razón comparamos en set de un volumen de 400 y 2000 periodistas el tiempo de ejecución cuando la cantidad mínima de jugadores era 3,4 y 5 respectivamente.

Cantidad de	Cantidad mínima de jugadores por periodista	Tiempo Ejecución Backtracking (s)	Tiempo Ejecucion PL (s)
400	3	0,20423	0,600313
400	4	0,78962	2,15414
400	5	2,76411	5,74428
2000	3	20,1419	18,2952
2000	4	164,192	37,43
2000	5	390,318	58,87

Figura 1: Resultados Bactracking vs Programación Lineal Entera

En la tabla podemos observar como aumenta el tiempo de ejecución cuando la cantidad mínima de jugadores elegidos por periodista incrementa.

5.6. Modelo de programación lineal con valores reales

Realizamos un modelo de Programación Lineal para el Hitting-Set Problem, permitiendo variables de decisión continuas para representar la inclusión de jugadores en el conjunto. Redondeamos la solución relajada asignando 1 a las variables de decisión de jugadores cuyo valor es mayor o igual a $1/b$, donde b es la cantidad de jugadores en el conjunto más grande entre los pedidos de la prensa.

Para demostrar qué tan buena es la aproximación, definimos:

$$\frac{A(I)}{z(I)} \leq r(A)$$

donde I es una instancia del HS Problem, $z(I)$ es una solución óptima para dicha instancia y $A(I)$ es la solución aproximada. Nuestro objetivo es encontrar la cota $r(A)$.

Para esto, vamos a definir w_i como el costo de utilizar al elemento i y queremos minimizar $\sum_{i=1}^{n} w_i x_i$ con la reestricción de que la suma de las variables $x_1, .., x_j$ de cada subconjunto B_i sea mayor igual a 1 con $0 \leq x_i \leq 1$. Esta metodología siempre encuentra un valor de la función objetivo mejor o igual a la solución óptima del HS. Es decir, la solución por programación lineal, llamémosle $z_{LP}(I)$ es $\leq z(I)$.

Sin embargo, para que el resultado sea válido vamos a redondear: una variable se traduce a 1 si es mayor o igual a $1/b$. Es decir, será parte de la solución todo elemento con coeficiente mayor o igual a $1/b$. Con esto nos aseguramos que para cada ecuación al menos un elemento se encienda.

Entonces, Programación Lineal "paga" $1/b$ o más por w_i pero nuestra solución $A(I)$ paga w_i (completo), es decir, $z_{LP}(I)$ paga $1/b$ o más por cada uno de los elementos que considera en el HS, mientras que $A(I)$ paga 1. Por lo tanto, nuestro óptimo es $1/b$ del óptimo de Programación Lineal, esto quiere decir que nuestro aproximado es como mucho b veces peor que $z(I)$:

$$A(I) = \sum_{a_i \in I} w_i \leq \sum_{a_i \in I} w_i \cdot bx_i \leq b \sum_{i=1}^{n} w_i x_i = b z_{LP}(I)$$

Por lo que concluímos que:

$$\frac{A(I)}{z(I)} \leq b$$

5.6.1. Complejidad

El código utiliza PuLP, una biblioteca de optimización lineal para Python, para resolver un problema de programación lineal. La parte principal del código que afecta la complejidad es la creación y resolución del problema de optimización. La creación de variables y restricciones tiene una complejidad que depende del número de variables y restricciones, y la resolución del problema puede tener una complejidad asociada con el método de solución Simplex. Al ser un problema lineal, los óptimos van a estar sí o sí en vértices. El algoritmo Simplex es en sí greedy: busca óptimos locales esperando que sean el óptimo general también. Su complejidad es $O(\text{Cantidad de vértices}) \rightarrow$ en promedio $O(n)$.

5.6.2. Mediciones

Realizamos mediciones utilizando el algoritmo exacto y la aproximación, con el objetivo de verificar la cota obtenida.

Archivo	Cantidad mínima esperada	Cantidad mínima obtenida	Se obtuvo el resultado esperado?	Tiempo de ejecución (s)
5.txt	2	2	Sí	0,02134823799133
7.txt	2	2	Sí	0,02002882957458
10_pocos.txt	3	6	No	0,01259016990662
10_varios.txt	6	6	Sí	0,01390266418457
10_todos.txt	10	10	Sí	0,01550531387329
15.txt	4	11	No	0,01595520973206
20.txt	5	12	No	0,01963591575623
50.txt	6	13	No	0,02509188652039
75.txt	8	20	No	0,02255940437317
100.txt	9	17	No	0,03087306022644
200.txt	9	24	No	0,03628802299500

Figura 2: Comparación de los resultados obtenidos por PL con los óptimos para los sets brindados por la cátedra

Archivo	Óptimo	Aproximación	b	A(I)/z(I)	¿Cumple relación de aproximación?
5.txt	2	2	4	1,00	Sí
7.txt	2	2	6	1,00	Sí
10_pocos.txt	3	6	7	2,00	Sí
10_varios.txt	6	6	3	1,00	Sí
10_todos.txt	10	10	4	1,00	Sí
15.txt	4	11	6	2,75	Sí
20.txt	5	12	7	2,45	Sí
50.txt	6	13	7	2,17	Sí
75.txt	8	20	8	2,45	Sí
100.txt	9	17	5	1,89	Sí
200.txt	9	24	7	2,67	Sí

Figura 3: Relación de la aproximación

Podemos concluír que la cota a la que llegamos $r(A) = b$ se cumple para todos los casos. Sin embargo, la aproximación no es lo suficiente buena, dado que para más de la mitad de los casos probados supera al óptimo en más de la mitad.

5.7. Algoritmo Greedy

Como bien sabemos, una solución Greedy se basa en tomar la decisión que nos brinda el óptimo local en cada paso. Es un hecho que este tipo de soluciones no siempre llegan a una solución óptima, pero si son buenas aproximaciones, principalmente por su velocidad. Para obtener una solución Greedy, consideramos que en cada paso, debemos tomar al jugador que a más periodistas deja satisfechos.

5.7.1. Implementación del Algoritmo

En primer lugar, debemos ordenar a los jugadores de manera decreciente, teniendo en cuenta su frecuencia en los pedidos de los periodistas, es decir, del más pedido al menos. Para esto creamos la siguiente función:

```
    def ordenar_jugadores_por_apariciones(pedidos_de_la_prensa):
        jugadores = [jugador for lista in pedidos_de_la_prensa for jugador in lista
    ]
        contador_jugadores = Counter(jugadores)
        jugadores_ordenados = sorted(contador_jugadores, key=lambda jugador:
    contador_jugadores[jugador], reverse=True)
        return jugadores_ordenados
```

Luego debemos añadir a nuestra solución el primer elemento de ese ordenamiento (es decir, el jugador más frecuente), para así ir generando una solución con los más pedidos. Esto debe estar hecho en un ciclo que itere hasta encontrar una solución que sea válida. Además cada vez que agregamos un jugador, actualizamos los pedidios de los periodistas quedandonos con aquellos que no fueron todavía satisfechos. Al hacer esta actualización, deja de ser válido el previo ordenamiento de los jugadores, por lo que debemos volver a ordenar con los periodistas restantes a satisfacer.

```
    jugadores_ordenados = ordenar_jugadores_por_apariciones(pedidos_de_la_prensa)
    while not solucionado:
        jugador_actual = jugadores_ordenados[0]
        solucion.append(jugador_actual)

        if verificar_solucion_para_Scaloni(constante, solucion):
            solucionado = True
        else:
            pedidos_de_la_prensa = [lista for lista in pedidos_de_la_prensa if
    jugador_actual not in lista]
            jugadores_ordenados = ordenar_jugadores_por_apariciones(
    pedidos_de_la_prensa)
```

5.7.2. Comparación con Aproximación LP

Realizamos una tabla en donde comparamos los resultados obtenidos por el algoritmo Greedy con el algoritmo redondeado de Programación Lineal, para los mismos sets de datos. Observamos que Greedy es una mejor aproximación.

Archivo	Resultado LP	Resultado Greedy	Cantidad Mínima Esperada
5.txt	2	2	2
7.txt	2	2	2
10_pocos.txt	6	3	3
10_varios.txt	6	7	6
10_todos.txt	10	10	10
15.txt	11	5	4
20.txt	12	5	5
50.txt	13	7	6
75.txt	20	9	8
100.txt	17	11	9
200.txt	24	10	9

Figura 4: Greedy vs Programación Lineal

5.7.3. Complejidad

Para determinar la complejidad vamos a analizar cada paso del algoritmo:

La función que ordena los jugadores crea una lista 'jugadores' que contiene todos los jugadores de los pedidos de la prensa. Esto toma $O(m \cdot b)$ donde m es el número de pedidos de la prensa y b es el máximo número de jugadores en un pedido. Luego, se utiliza la función 'Counter' para contar las apariciones de cada jugador. Esto toma $O(m \cdot b)$. Finalmente, se ordena la lista de jugadores por apariciones utilizando 'sorted', lo cual tiene una complejidad de $O(n \log n)$, donde n es el número total de jugadores únicos. En total, la complejidad de 'ordenar_jugadores_por_apariciones' sería aproximadamente $O(m \cdot b + n \log n)$.

Por otro lado, el algoritmo greedy en sí, llama a 'ordenar_jugadores_por_apariciones', por lo que contribuye con la complejidad mencionada anteriormente. También utiliza un bucle 'while' que puede iterar hasta n veces (el número de jugadores únicos). En cada iteración, realiza operaciones que toman $O(m \cdot b)$ debido a la verificación de la solución con 'verificar_solucion_para_Scaloni' y la manipulación de la lista de pedidos de la prensa.

La complejidad total de 'solucion_greedy' sería aproximadamente $O(n \cdot m \cdot b)$, donde n es el número de jugadores únicos, m es el número de pedidos de la prensa y b es el máximo número de jugadores en un pedido.

6. Conclusiones

En conclusión, en este trabajo abordamos diversos aspectos relacionados con la programación lineal y la resolución de problemas de optimización, asi como también soluciones menos eficientes como lo es backtracking.

A lo largo de este informe, se destacó la importancia de considerar la complejidad algorítmica en el diseño de algoritmos, especialmente cuando se trabaja con conjuntos de datos de gran escala. Se exploraron dos aproximaciones que si bien no son siempre óptimas, son muy rápidas y pueden servir como un acercamiento a la solución.

Podríamos decir que logramos un algoritmo de backtracking que resuelve el problema de Scaloni de manera óptima. Al momento de realizar pruebas con los sets de la cátedra vimos que obtuvimos los resultados esperados en poco tiempo. Sin embargo, al aumentar la cantidad de periodistas por set, o al probar con sets propios donde modificabamos la cantidad mínima de jugadores por periodista, la complejidad temporal aumentaba notablemente. El crecimiento es más notorio cuando se aumenta la cantidad mínima por cada periodista, tal que un set de datos con 300k de periodistas se resolvia "más rápido"que uno con 2000 periodistas pero con la restricción de que la cantidad mínima de jugadores elegidos sea ¿1. Esto ocurre porque al tener 300k periodistas con su pedido generado aleatoriamente, se generaban pedidos de largo 1 con un jugador, y así pasaba con todos los jugadores disponibles. Esto generaba que cuando nuestro algoritmo ordena los sets por su longitud, el algoritmo recorría todos esos conjuntos de largo 1 y generaba una solución con todos los jugadores posibles.

Por programación lineal entera logramos encontrar soluciones óptimas en mejor tiempo que con Backtracking. Para este problema podría decirse que es el algoritmo que mejor resuelve.

Si permitimos que nuestras variables tomen valores reales, vemos que obtenemos una aproximación que está lejos de ser buena en comparación a la obtenida por el algoritmo Greedy implementado.

Por lo tanto, dependiendo el set de datos al que nos enfrentamos, debemos de analizar previamente el mismo y decidir con qué algoritmo vamos a resolverlo. Si tenemos mucho tiempo quizas nos inclinamos por Backtracking o Programación Lineal ya que ambas aseguran la solución óptima pero a costo de una mayor complejidad temporal. En cambio, si necesitamos una aproximación rápida y no necesariamente óptima, inclinarse por un algoritmo Greedy seria una buena elección.